Type-theoretic extensions of Abstract Categorial Grammars

Philippe De Groote, Sarah Maarek

Type-theoretic extensions of Abstract Categorial Grammars

Philippe de Groote[1] and Sarah Maarek[2]

[1] LORIA & INRIA-Lorraine
Philippe.de.Groote@loria.fr

[2] LORIA & Université Nancy 2
Sarah.Maarek@loria.fr

1 Introduction

Abstract Categorial Grammars (ACG), in their original definition [3], are based on the linear λ-calculus. This choice derives from the traditional categorial grammars, which are based on resource sensitive logics [8].

From a language-theoretic standpoint, this linearity constraint does not result in a weak expressive power of the formalism [4, 5]. In particular, the string languages generated by the second-order ACGs (whose parsing is known to be polynomial [12]) corresponds to the class of mildly context sensitive languages. From a more practical point of view, however, it would be interesting to increase the intentional expressive power of the formalism by providing high level constructs. For instance, one would like to provide the ACGs with feature structures, as it is the case in most current grammatical formalisms.

In [3], a possible way of extending the ACGs is proposed. It consists of enriching the type system of the formalism with new type constructors. The present paper, which elaborates on this proposal, is organized as follows:

- In the next section, we remind the reader of the definition of an ACG. Then, we explain why the ACG architecture is well-suited for type-theoretic extensions.
- In Section 3, we briefly review and motivate possible extensions based on the following type constructors: non-linear functional types, cartesian product, disjoint union, unit type, and dependent product.
- In Section 4, we define formally the type system underlying the extensions proposed in Section 3.
- Finally, we illustrate the resulting system by providing a toy example.

2 The ACG architecture

An Abstract Categorial Grammar consists of two signatures together with a morphism that allows the types and the terms built on the first signature to be interpreted as types and terms built on the second signature. This morphism

(which is called the *lexicon* of the grammar) is required to commute with the typing relation, which ensures that well-typed terms are interpreted as well-typed terms.

More formally, let $\mathscr{T}(A)$ denotes the set of linear functional types[1] built from the set of atomic types A, and define a higher-order linear signature Σ to be a triple $\langle A, C, \tau \rangle$, where A is a finite set of atomic types, C is a finite set of constants, and $\tau : C \to \mathscr{T}(A)$ is a function that assigns a linear functional type to each constant. Given such a signature Σ, define $\Lambda(\Sigma)$ to be the set of (well-typed) linear λ-terms built on Σ. Then, given two higher-order linear signatures $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$, define a lexicon from Σ_1 to Σ_2 to be a pair $\mathscr{L} = \langle \eta, \theta \rangle$ where $\eta : A_1 \to \mathscr{T}(A_2)$ and $\theta : C_1 \to \Lambda(\Sigma_2)$ are such that:

$$\vdash_{\Sigma_2} \theta(c) : \eta(\tau_1(c)) \text{ for all } c \in C_1$$

This condition ensures that the homomorphic extensions of η and θ are such that well-typed terms are interpreted by well-typed terms.

Using the above definitions,, an abstract categorial grammar is defined to be a quadruple $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, s \rangle$ where:

1. Σ_1 and Σ_2 are two higher-order linear signatures;
2. $\mathscr{L}$ is a lexicon form Σ_1 to Σ_2;
3. s is an atomic type of Σ_1 called the *distinguished type* of the grammar.

From a methodological point of view, the first signature (which is called the *abstract vocabulary*) is used to specify the abstract parse structures of the grammar, while the second signature (which is called the *object vocabulary*) is used to express the surface forms of the grammar.[2] More precisely, an Abstract Categorial Grammar generates two languages: an abstract language, which corresponds to Curry's tectogrammatics, and an object language, which corresponds to Curry's phenogrammatics [2]. From a formal point of view, the abstract language is simply defined to be the set of closed terms, built on the abstract vocabulary, that are of the given distinguished type:

$$\mathcal{A}(\mathscr{G}) = \{t \in \Lambda(\Sigma_1) \mid \vdash_{\Sigma_1} t : s\}$$

Then, the object language is defined to be the image of the abstract language under the lexicon:

$$\mathcal{O}(\mathscr{G}) = \{t \in \Lambda(\Sigma_2) \mid \exists u \in \mathcal{A}(\mathscr{G}).\ t = \theta(u)\}$$

It is important to note that this architecture is in fact independent of the underlying type system. This provides us with a systematic way of extending the expressive power of the ACG framework, simply by enriching the underlying type theory.

[1] We use the connective $\multimap$ —i.e., the linear implication symbol— to denote the linear functional type constructor.

[2] It may also be the case that the object vocabulary is used to express the logical forms of the grammar. In the present setting, the notions of syntax, semantics, parse structure, surface form, logical form, ... are purely methodological. These notions do not belong to the theory of ACGs.

3 The need for type-theoretic extensions

In [3], we suggest that all the connectives of linear logic (seen, through the Curry-Howard isomorphism, as type constructors) are natural candidates for extending the ACG typing system. In this paper, we propose extensions based on the following type constructors:

- Non-linear functional type. This corresponds, by the Curry-Howard isomorphism, to intuitionistic implication. It is not a primitive connective of linear logic, but may be defined by means of the linear implication and the "of course" modality.
- Cartesian product, disjoint union, and unit type. These corresponds to the additive conjunction and disjunction, and the multiplicative unit.
- Dependent product. This type construct does not correspond to any linear logic primitive. It comes from Matin-Löf type-theory [7] and has been used in a linear logic context by Pfenning and Cervesato [1].

This choice is not a matter of doctrine. It rather results from experiments in writing toy grammars.

Non-linearity. Categorial grammars are based on linear type-theories because grammatical composition has been identified as a resource-sensitive process. This does not mean, however, that grammatical composition is always linear. It rather means that categorial type logics should allow for linearity.

Non linear combinators are quite usual in semantics. For instance, the lexical semantics of an intersective adjective such as *red* is given by the following non-linear λ-term:[3]

$$\lambda^{\circ} P.\, \lambda x.\, (P\, x) \wedge (\mathbf{red}\, x) : (\iota \to o) \multimap (\iota \to o) \tag{1}$$

Another typical use of non-linearity is provided, on the syntactic side, by feature agreement. This is well exemplified in Ranta's GF [11].

In the ACG framework, both syntax and semantics are modeled using the same primitives. As a consequence, a λ-term such as (1) must be allowed as a possible lexical entry. Consequently, there is a crucial need for non-linear λ-abstraction.

From a technical point of view, mixing linear and non-linear functional types results in a typing system with two kinds of contexts: linear contexts, and non-linear contexts. Such a system will be defined in Section 4.

Cartesian product, disjoint union, and unit. As we already mentioned in the introduction of this paper, we feel a need for providing ACGs with feature structures. These may be defined using records, variants, and enumerated types

[3] We write $\lambda^{\circ} x.\, t$ for a linear λ-abstraction, and $\lambda x.\, t$ for a non-linear (i.e., usual) λ-abstraction.

(which may appear to be particular cases of variants). For instance, the following feature matrix:

$$\begin{bmatrix} \text{inflection} = \begin{bmatrix} \text{gender} & = \text{masc} \\ \text{number} & = \text{sing} \end{bmatrix} \\ \text{function} \quad = \text{object}\,(\text{indirect}\,(\text{a\_Prep})) \end{bmatrix}$$

may be expressed as a well-typed term of the following signature:[4]

$$\begin{aligned} \mathit{Gender} &= \{\text{masc} \mid \text{fem}\} : \text{type} \\ \mathit{Number} &= \{\text{sing} \mid \text{plur}\} : \text{type} \\ \mathit{Inflection} &= [\text{gender} : \mathit{Gender};\ \text{number} : \mathit{Number}] : \text{type} \\ \mathit{Preposition} &= \{\text{a\_Prep} \mid \text{de\_Prep}\} : \text{type} \\ \mathit{Direction} &= \{\text{direct} \mid \text{indirect } \texttt{of } \mathit{Preposition}\} : \text{type} \\ \mathit{Function} &= \{\text{subject} \mid \text{object } \texttt{of } \mathit{Direction}\} : \text{type} \\ \mathit{Features} &= [\text{inflection} : \mathit{Inflection};\ \text{function} : \mathit{Function}] : \text{type} \end{aligned}$$

From a type theoretic standpoint, records, variants, and enumerated types may be defined using cartesian products, disjoint unions, and unit types.

Dependent product Dependent product is a quite powerful type construction. It allows ones to specify types that depend upon terms. This is exactly what is needed if one wants to define generic syntactic categories (for instance, *NP* for *noun phrase*) that can be instantiated according to the value of some feature (for instance, $(NP\,f)$ for *feminine noun phrase*, $(NP\,m)$ for *masculine noun phrase*, etc.)

Dependent products are not that much popular in the type logical grammar community, even if some weak form of dependent function is mentioned by both Morrill [9] and Moortgat [8]. There is, however, quite an exception to this matter of fact, namely, Ranta's type-theoretical grammars [10]. If Ranta's Grammatical Framework is definitively an achievement that speaks for itself, it also speaks for the use of dependent products.

4 The extended typing system

In the presence of dependent products, the well-formedness of types may depend upon the well-typedness of terms. The usual solution to this is to consider a third level of expressions: the kinds. These are to the types what the types are to the terms. Consequently, the extended typing system we propose relies on four sorts of typing judgements that are defined by mutual induction:

- The judgement that a signature is well-formed.
- The judgement that a kind is well-formed.

[4] We use square brackets for records and curly brackets for variants. We hope that the reader will find this syntax self-explanatory. If this is not the case, we refer him/her to Section 4.

– The judgement that a type is well-kinded.
– The judgement that a term is well-typed.

These judgements manipulate expressions that obey the following raw syntax.

Signatures

$$
\begin{array}{lll}
\Sigma ::= () & & (\textit{Empty signature}) \\
\quad | \quad \Sigma;\, a : \mathsf{K} & & (\textit{Atomic type declaration}) \\
\quad | \quad \Sigma;\, a = \mathrm{R} : \mathsf{type} & & (\textit{Record type declaration}) \\
\quad | \quad \Sigma;\, a = \mathrm{V} : \mathsf{type} & & (\textit{Variant type declaration}) \\
\quad | \quad \Sigma;\, c : \tau & & (\textit{Constant declaration})
\end{array}
$$

Kinds

$$
\begin{array}{lll}
\mathsf{K} ::= \mathsf{type} & & (\textit{Kind of types}) \\
\quad | \quad (\tau)\mathsf{K} & & (\textit{Kind of dependent types})
\end{array}
$$

Types

$$
\begin{array}{lll}
\tau ::= a & & (\textit{Atomic type}) \\
\quad | \quad (\lambda x.\, \tau) & & (\textit{Type abstraction}) \\
\quad | \quad (\tau\, t) & & (\textit{Type application}) \\
\quad | \quad (\tau_1 \multimap \tau_2) & & (\textit{Linear functional type}) \\
\quad | \quad (\Pi x : \tau_1)\, \tau_2) & & (\textit{Dependent product})
\end{array}
$$

Record types

$$
\begin{array}{lll}
\mathrm{R} ::= [l_1 : \tau_1; \ldots; l_n : \tau_n] & & (\textit{Record type})
\end{array}
$$

Variant types

$$
\begin{array}{lll}
\mathrm{V} ::= \{c_1 \texttt{ of } \tau_1 | \ldots | c_n \texttt{ of } \tau_n\} & & (\textit{Variant type})
\end{array}
$$

Terms

$$
\begin{array}{lll}
t ::= c & & (\textit{Constant}) \\
\quad | \quad x & & (\textit{Variable}) \\
\quad | \quad (\lambda^{\circ} x.\, t) & & (\textit{Linear abstraction}) \\
\quad | \quad (\lambda x.\, t) & & (\textit{Non-linear abstraction}) \\
\quad | \quad (t_1\, t_2) & & (\textit{Application}) \\
\quad | \quad [l_1 = t_1; \ldots; l_n = t_n] & & (\textit{Record}) \\
\quad | \quad t.l & & (\textit{Selection}) \\
\quad | \quad \{(c_1\, x_1) \to t_1 \mid \ldots \mid (c_n\, x_n) \to u_n\} & & (\textit{Case analysis})
\end{array}
$$

In this grammar, a, c, x, and l (possibly with subscripts) range over type constants, λ-term constants,[5] λ-variables, and record labels, respectively. In a variant type the "$\texttt{ of } \tau_i$" part is optional. In which case, the corresponding variant

[5] Variant constructors are considered as a special case of constants.

constructor c_i amounts to a constant. This allows enumerated types to be seen as particular cases of variants.

Given a (raw) signature Σ, we define three partial functions. The first one, $kind_\Sigma$, takes a type constant as an argument and possibly yields a kind as a result. It is inductively defined as follows:

$$
\begin{aligned}
kind_{()}(a) &\text{ is undefined} \\
kind_{\Sigma;\, a_1:\mathsf{K}}(a) &= \begin{cases} \mathsf{K} & \text{if } a = a_1 \\ kind_\Sigma(a) & \text{otherwise} \end{cases} \\
kind_{\Sigma;\, a_1=\mathrm{R}:\mathsf{type}}(a) &= \begin{cases} \mathsf{type} & \text{if } a = a_1 \\ kind_\Sigma(a) & \text{otherwise} \end{cases} \\
kind_{\Sigma;\, a_1=\mathrm{V}:\mathsf{type}}(a) &= \begin{cases} \mathsf{type} & \text{if } a = a_1 \\ kind_\Sigma(a) & \text{otherwise} \end{cases} \\
kind_{\Sigma;\, c:\tau}(a) &= kind_\Sigma(a)
\end{aligned}
$$

Similarly, $type_\Sigma$ assigns types to λ-term constants:

$$
\begin{aligned}
type_{()}(c) &\text{ is undefined} \\
type_{\Sigma;\, a_1:\mathsf{K}}(c) &= type_\Sigma(c) \\
type_{\Sigma;\, a_1=\mathrm{R}:\mathsf{type}}(c) &= type_\Sigma(c) \\
type_{\Sigma;\, a_1=\mathrm{V}:\mathsf{type}}(c) &= type_\Sigma(c) \\
type_{\Sigma;\, c_1:\tau}(c) &= \begin{cases} \tau & \text{if } c = c_1 \\ type_\Sigma(c) & \text{otherwise} \end{cases}
\end{aligned}
$$

Finally, $binding_\Sigma$, takes a type constant as an argument and possibly yields a record or variant type:

$$
\begin{aligned}
binding_{()}(a) &\text{ is undefined} \\
binding_{\Sigma;\, a_1:\mathsf{K}}(a) &= \begin{cases} \text{undefined} & \text{if } a = a_1 \\ binding_\Sigma(a) & \text{otherwise} \end{cases} \\
binding_{\Sigma;\, a_1=\mathrm{R}:\mathsf{type}}(a) &= \begin{cases} \mathrm{R} & \text{if } a = a_1 \\ binding_\Sigma(a) & \text{otherwise} \end{cases} \\
binding_{\Sigma;\, a_1=\mathrm{V}:\mathsf{type}}(a) &= \begin{cases} \mathrm{V} & \text{if } a = a_1 \\ binding_\Sigma(a) & \text{otherwise} \end{cases} \\
binding_{\Sigma;\, c:\tau}(a) &= binding_\Sigma(a)
\end{aligned}
$$

As we already said, the rules of the typing system involve four sorts of judgements. They are of the following forms:

$\mathsf{sig}\,(\Sigma)$	(Σ *is a well-formed signature*)
$\vdash_\Sigma \mathsf{K} : \mathsf{kind}$	(*Given the signature* Σ, K *is a well-formed kind*)
$\Gamma \vdash_\Sigma \alpha : \mathsf{K}$	(*Given the signature* Σ, α *is a type of kind* K *according to the non-linear typing context* Γ)
$\Gamma; \Delta \vdash_\Sigma t : \alpha$	(*Given the signature* Σ, t *is a term of type* α *according to the non-linear typing context* Γ *and the linear typing context* Δ)

We are now in a position of giving the very rules of the typing system.

Well-formed signatures

$$\mathsf{sig}\,(\,)$$

$$\frac{\mathsf{sig}\,(\Sigma) \qquad \vdash_\Sigma \mathsf{K} : \mathsf{kind}}{\mathsf{sig}\,(\Sigma;\, a : \mathsf{K})}$$

$$\frac{\mathsf{sig}\,(\Sigma) \qquad \vdash_\Sigma \alpha_1 : \mathsf{type} \qquad \cdots \qquad \vdash_\Sigma \alpha_n : \mathsf{type}}{\mathsf{sig}\,(\Sigma;\, a = [l_1 : \alpha_1; \ldots; l_n : \alpha_n] : \mathsf{type})}$$

$$\frac{\mathsf{sig}\,(\Sigma) \qquad \vdash_\Sigma \alpha_1 : \mathsf{type} \qquad \cdots \qquad \vdash_\Sigma \alpha_n : \mathsf{type}}{\mathsf{sig}\,(\Sigma;\, a = \{c_1 \ \mathtt{of}\ \alpha_1 | \ldots | c_n \ \mathtt{of}\ \alpha_n\} : \mathsf{type})}$$

$$\frac{\mathsf{sig}\,(\Sigma) \qquad \vdash_\Sigma \alpha : \mathsf{type}}{\mathsf{sig}\,(\Sigma;\, c : \alpha)}$$

In the above rules, all the introduced symbols (a, $l_1, \ldots, l_n$, $c_1, \ldots, c_n$, c) must be fresh with respect to Σ.

Well-formed kinds

$$\vdash_\Sigma \mathsf{type} : \mathsf{kind}$$

$$\frac{\vdash_\Sigma \alpha : \mathsf{type} \qquad \vdash_\Sigma \mathsf{K} : \mathsf{kind}}{\vdash_\Sigma (\alpha)\,\mathsf{K} : \mathsf{kind}}$$

Well-kinded types

$$\vdash_\Sigma a : \mathit{kind}_\Sigma(a) \quad \text{(type const.)}$$

$$\frac{\vdash_\Sigma \alpha : \mathsf{type} \quad \Gamma \vdash_\Sigma \beta : \mathsf{K}}{\Gamma, x : \alpha \vdash_\Sigma \beta : \mathsf{K}} \quad \text{(type weak.)}$$

$$\frac{\Gamma, x : \alpha \vdash_\Sigma \beta : \mathsf{K}}{\Gamma \vdash_\Sigma \lambda x.\beta : (\alpha)\,\mathsf{K}} \quad \text{(type abs.)}$$

$$\frac{\Gamma \vdash_\Sigma \alpha : (\beta)\,\mathsf{K} \quad \Gamma;\ \vdash_\Sigma t : \beta}{\Gamma \vdash_\Sigma \alpha\, t : \mathsf{K}} \quad \text{(type app.)}$$

$$\frac{\Gamma \vdash_\Sigma \alpha : \mathsf{type} \quad \Gamma \vdash_\Sigma \beta : \mathsf{type}}{\Gamma \vdash_\Sigma \alpha \multimap \beta : \mathsf{type}} \quad \text{(lin. fun.)}$$

$$\frac{\Gamma \vdash_\Sigma \alpha : \mathsf{type} \quad \Gamma, x : \alpha \vdash_\Sigma \beta : \mathsf{type}}{\Gamma \vdash_\Sigma (\Pi x : \alpha)\,\beta : \mathsf{type}} \quad \text{(dep. prod.)}$$

In Rule (type weak.), x must be fresh with respect to Γ.

Well-typed terms

$$;\ \vdash_\Sigma c : \mathit{type}_\Sigma(c) \quad \text{(const.)}$$

$$\frac{\Gamma \vdash_\Sigma \alpha : \mathsf{type}}{\Gamma;\ x : \alpha \vdash_\Sigma x : \alpha} \quad \text{(lin. var.)}$$

$$\frac{\Gamma \vdash_\Sigma \alpha : \mathsf{type}}{\Gamma,\ x : \alpha;\ \vdash_\Sigma x : \alpha} \quad \text{(var.)}$$

$$\frac{\Gamma \vdash_\Sigma \alpha : \mathsf{type} \quad \Gamma;\ \Delta \vdash_\Sigma t : \beta}{\Gamma, x : \alpha;\ \Delta \vdash_\Sigma t : \beta} \quad \text{(weak.)}$$

$$\frac{\Gamma;\ \Delta_1, x : \alpha, y : \beta, \Delta_2 \vdash_\Sigma t : \gamma}{\Gamma;\ \Delta_1, y : \beta, x : \alpha, \Delta_2 \vdash_\Sigma t : \gamma} \quad \text{(perm.)}$$

$$\frac{\Gamma;\ \Delta, x : \alpha \vdash_\Sigma t : \beta}{\Gamma;\ \Delta \vdash_\Sigma \lambda^\circ x.t : \alpha \multimap \beta} \quad \text{(lin. abs.)}$$

$$\frac{\Gamma;\ \Delta_1 \vdash_\Sigma\ t : \alpha \multimap \beta \quad \Gamma;\ \Delta_2 \vdash_\Sigma\ u : \alpha}{\Gamma;\ \Delta_1, \Delta_2 \vdash_\Sigma\ t\,u : \beta} \quad \text{(lin. app.)}$$

$$\frac{\Gamma, x : \alpha;\ \Delta \vdash_\Sigma\ t : \beta}{\Gamma;\ \Delta \vdash_\Sigma\ \lambda x.\, t : (\Pi x : \alpha)\,\beta} \quad \text{(abs.)}$$

$$\frac{\Gamma;\ \Delta \vdash_\Sigma\ t : (\Pi x : \alpha)\beta \quad \Gamma;\ \vdash_\Sigma\ u : \alpha}{\Gamma;\ \Delta \vdash_\Sigma\ t\,u : \beta[x{:=}u]} \quad \text{(app.)}$$

$$\frac{\mathit{binding}_\Sigma(a) = [l_1 : \alpha_1; \ldots; l_n : \alpha_n] \quad \Gamma;\ \Delta \vdash_\Sigma\ t_i : \alpha_i\ \ (1 \leq i \leq n)}{\Gamma;\ \Delta \vdash_\Sigma\ [l_1 = t_1; \ldots; l_n = t_n] : a} \quad \text{(rec.)}$$

$$\frac{\mathit{binding}_\Sigma(a) = [l_1 : \alpha_1; \ldots; l_n : \alpha_n] \quad \Gamma;\ \Delta \vdash_\Sigma\ t : a}{\Gamma;\ \Delta \vdash_\Sigma\ t.l_i : \alpha_i} \quad \text{(sel.)}$$

$$\frac{\mathit{binding}_\Sigma(a) = \{c_1\ \mathtt{of}\ \alpha_1 | \ldots | c_n\ \mathtt{of}\ \alpha_n\}}{;\vdash_\Sigma\ c_i : \alpha_i \multimap a} \quad \text{(inj.)}$$

$$\frac{\mathit{binding}_\Sigma(a) = \{c_1\ \mathtt{of}\ \alpha_1 | \ldots | c_n\ \mathtt{of}\ \alpha_n\} \quad \Gamma;\ \Delta, x_i : \alpha_i \vdash_\Sigma\ t_i : \beta\ \ (1 \leq i \leq n)}{\Gamma;\ \Delta \vdash_\Sigma\ \{(c_1\,x_1) \to t_1 \mid \ldots \mid (c_n\,x_n) \to t_n\} : a \multimap \beta} \quad \text{(case)}$$

In Rules (lin. var.) and (var.), x must be fresh with respect to Γ. In Rule (weak.), x must be fresh with respect to both Γ and Δ. Moreover, t must be either a λ-variable, a constant, or a variant constructor. In Rule (abs.), x cannot occur free in Δ.

Both the abstract and the object language of an ACG are defined modulo an appropriate notion of equality between λ-terms. In the original definition of an ACG, this notion of equality is the usual relation of $\beta\eta$-conversion. In the present setting, an equivalence relation akin to β-conversion may be defined as the reflexive, transitive, symmetric closure of the reduction relation induced by the following rules.

$$(\lambda^\circ x.\, t)\, u \to t[x{:=}u] \qquad (\textit{Linear } \beta\textit{-reduction})$$
$$(\lambda x.\, t)\, u \to t[x{:=}u] \qquad (\beta\textit{-reduction})$$
$$[l_1 = t_1; \ldots; l_n = t_n].l_i \to t_i \qquad (\textit{Record selection})$$
$$\{(c_1\,x_1) \to t_1 \mid \ldots \mid (c_n\,x_n) \to t_n\}\,(c_i\,u) \to t_i[x_i{:=}u] \qquad (\textit{Case analysis})$$

This reduction relation satisfies the properties of confluence, subject reduction, and strong normalization. Hence, the corresponding equivalence relation is

decidable. Nevertheless, it amounts to a rather weak form of equality. In order to obtain a stronger notion of equality, one should also consider η-like conversion rules and permutative conversion rules. We will not discuss these in the present paper.

Other theoretical questions concern the decidability and the tractability of parsing. It is not difficult to show that ACGs based on this extended typing system have an undecidable membership problem. This follows from the fact that the Edinburgh logical framework [6] appear to be a subsystem of the present typing system.[6] This raises the problem of isolating interesting fragments that have a decidable membership problem.

5 A small example

In order to illustrate the expressive power of the extensions we have introduced, we provide a toy example related to gender agreement in French. This example implements the rule saying that, in the plural, the masculine is used by default when referring to a group of mixed genders.

We first give an abstract signature:

$$
\begin{aligned}
\mathit{gender} &= \{\mathrm{m} \mid \mathrm{f}\} : \mathrm{type} \\
\mathit{number} &= \{\mathrm{s} \mid \mathrm{p}\} : \mathrm{type} \\
\mathit{m\_feat} = [\mathrm{g} : \mathit{gender};\ \mathrm{n} : \mathit{number}] &: \mathrm{type} \\
N, NP &: (\mathit{m\_feat})\mathrm{type} \\
S &: \mathrm{type} \\
\textsc{pierre},\ \textsc{jean} &: NP[\mathrm{g} = \mathrm{m};\ \mathrm{n} = \mathrm{s}] \\
\textsc{marie},\ \textsc{alice} &: NP[\mathrm{g} = \mathrm{f};\ \mathrm{n} = \mathrm{s}] \\
\textsc{etre} &: (\Pi x : \mathit{m\_feat}) \\
&\qquad ((\Pi y : \mathit{m\_feat})(N\ y) \multimap NP\ x \multimap S) \\
\textsc{et} &: (\Pi xy : \mathit{m\_feat}) \\
&\qquad (NP\ x \multimap NP\ y \multimap NP[\mathrm{g} = C\ x\ y;\ \mathrm{n} = \mathrm{p}]) \\
\textsc{mathematicien} &: (\Pi x : \mathit{m\_feat}) N\ x
\end{aligned}
$$

where

$$C\ x\ y = \{\mathrm{m} \to \mathrm{m} \mid \mathrm{f} \to \{\mathrm{m} \to \mathrm{m} \mid \mathrm{f} \to \mathrm{f}\}\ (y.\mathrm{g})\}\ (x.\mathrm{g})$$

is the term specifying that the gender of a conjunction is feminine if and only if both conjuncts are of feminine gender.

[6] Actually, the notion of dependent type we use is slightly weaker. This, however, does not affect the undecidability result

We then consider the following object signature:

$$\begin{aligned} \mathit{gender} &= \{\mathrm{m} \mid \mathrm{f}\} : \mathrm{type} \\ \mathit{number} &= \{\mathrm{s} \mid \mathrm{p}\} : \mathrm{type} \\ \mathit{m\_feat} &= [\mathrm{g} : \mathit{gender}; \mathrm{n} : \mathit{number}] : \mathrm{type} \\ s &: \mathrm{type} \\ /\mathrm{Pierre}/,\ /\mathrm{Jean}/,\ /\mathrm{Marie}/,\ /\mathrm{Alice}/,\ /\mathrm{est}/,\ /\mathrm{sont}/,\ /\mathrm{et}/,& \\ /\mathrm{mathématicien}/,\ /\mathrm{mathématicienne}/,& \\ /\mathrm{mathématiciens}/,\ /\mathrm{mathématiciennes}/ &: s \multimap s \end{aligned}$$

Finally, we define the following lexicon:[7]

$$\begin{aligned} \mathit{gender} &:= \mathit{gender} \\ \mathit{number} &:= \mathit{number} \\ \mathit{m\_feat} &:= \mathit{m\_feat} \\ N, NP &:= \lambda x.\, s \multimap s \\ S &:= s \multimap s \\ \textsc{pierre} &:= /\mathrm{Pierre}/ \\ \textsc{jean} &:= /\mathrm{Jean}/ \\ \textsc{marie} &:= /\mathrm{Marie}/ \\ \textsc{alice} &:= /\mathrm{Alice}/ \\ \textsc{etre} &:= \lambda m.\, \lambda^{\circ} xy.\, y + (\{\, \mathrm{s} \to /\mathrm{est}/ \mid \mathrm{p} \to /\mathrm{sont}/ \,\}\, m.\mathrm{n}) + x\, m \\ \textsc{et} &:= \lambda m_1 m_2.\, \lambda^{\circ} xy.\, x + /\mathrm{et}/ + y \\ \textsc{mathematicien} &:= \lambda m.\, \{\, \mathrm{m} \to \{\, \mathrm{s} \to /\mathrm{mathématicien}/ \\ &\qquad\qquad \mid \mathrm{p} \to /\mathrm{mathématiciens}/ \,\}\, m.\mathrm{n} \\ &\qquad \mid \mathrm{f} \to \{\, \mathrm{s} \to /\mathrm{mathématicienne}/ \\ &\qquad\qquad \mid \mathrm{p} \to /\mathrm{mathématiciennes}/ \,\}\, m.\mathrm{n} \,\}\, m.\mathrm{g} \end{aligned}$$

Let us write [ms], [mp], [fs], and [fp] as abbreviations for [g = m; n = s], [g = m; n = p], [g = f; n = s], [g = f; n = p], respectively. We then have that the following terms are well-typed λ-terms belonging to the abstract language of the grammar:

$$\begin{aligned} &\textsc{etre}\ [\mathrm{ms}]\ \textsc{mathematicien}\ \textsc{pierre} \\ &\textsc{etre}\ [\mathrm{fp}]\ \textsc{mathematicien}\ (\textsc{et}\ [\mathrm{fs}]\ [\mathrm{fs}]\ \textsc{marie}\ \textsc{alice}) \\ &\textsc{etre}\ [\mathrm{mp}]\ \textsc{mathematicien}\ (\textsc{et}\ [\mathrm{fs}]\ [\mathrm{ms}]\ \textsc{marie}\ \textsc{pierre}) \end{aligned}$$

Their images by the lexicon yield respectively the following object terms:

$$\begin{aligned} &/\mathrm{Pierre}/ + /\mathrm{est}/ + /\mathrm{mathématicien}/ \\ &/\mathrm{Marie}/ + /\mathrm{et}/ + /\mathrm{Alice}/ + /\mathrm{sont}/ + /\mathrm{mathématiciennes}/ \\ &/\mathrm{Marie}/ + /\mathrm{et}/ + /\mathrm{Pierre}/ + /\mathrm{sont}/ + /\mathrm{mathématiciens}/ \end{aligned}$$

[7] As usual, strings are identified with λ-terms of type $s \multimap s$. Then, in case t and u are strings, $t + u$ stands for $\lambda x.\, t\,(u\, x)$.

References

1. Cervesato, I. and Pfenning, F. A linear logical framework. *Information & Computation*, 179(1): 19–75, 2002.
2. Curry, H.B. Some logical aspects of grammatical structure. In: *Proceedings of symposia in applied mathematics*, volume XII, pp. 56–68. 1961.
3. de Groote, Ph. Towards abstract categorial grammars. In: *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pp. 148–155, 2001.
4. de Groote, Ph. Tree-Adjoining Grammars as Abstract Categorial Grammars. In: *TAG+6, Proceedings of the sixth International Workshop on Tree Adjoining Grammars and Related Frameworks*, pp. 145–150, 2001.
5. de Groote, Ph. and Pogodalla, S. On the Expressive Power of Abstract Categorial Grammars: Representing Context-Free Formalisms. *Journal of Logic, Language and Information* 13(4): 421–438, 2004.
6. Harper, R., Honsel, F., and Plotkin, G. A framework for defining logics *Proceedings of the second annual IEEE symposium on logic in computer science* 194–204, 1987.
7. Martin-Löf, P. An Intuitionistic Theory of Types: Predicative Part. In: *Logic Colloquium '73*, North-Holland, pp. 73–118, 1975.
8. Moortgat, M. Categorial type logic. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, chapter 2. Elsevier, 1997.
9. Morrill, G. *Type Logical Grammar: Categorial Logic of Signs.* Kluwer Academic Publishers, 1994.
10. Ranta, A. *Type theoretical grammar.* Oxford University Press, 1994.
11. Ranta, A. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2): 145–189, 2004.
12. Salvati, S. *Problèmes de filtrage et problèmes d'analyse pour les grammaires catégorielles abstraites.* Thèse de Doctorat. Institut National Polytechnique de Lorraine, 2005